МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
к лабораторным работам

«Основы параллельного программирования на языке C#»

по дисциплине «Технологии программирования»

для студентов специальностей

122 – Компьютерные науки и информационные технологии,

124 – Системный анализ, в том числе для иностранных студентов

Утверждено
редакционно-издательским
советом университета,
протокол № 3 от 22.12.16 г

Харьков
НТУ «ХПИ»
2018

Методические указания к лабораторным работам «Основы параллельного программирования на языке C#» по дисциплине «Технологии программирования» : для студентов специальностей 122 – Компьютерные науки и информационные технологии, 124 – Системный анализ, в том числе для иностранных студентов / сост.: Ю. Н. Кожин, О. Н. Малых, В. Ф. Прокопенков. – Харьков : НТУ «ХПИ», 2018. – 64 с. – На рус. яз.

Составители: Ю. Н. Кожин,
О. Н. Малых,
В. Ф. Прокопенков,

Рецензент А. В. Горелый

Кафедра системного анализа и информационно-аналитических технологий

ВСТУПЛЕНИЕ

Современные многоядерные ПЭВМ представляют собой мощные вычислительные средства, и они используются неэффективно, если программное обеспечение для них представляют собой обычные последовательные программы. В системах с общей памятью, включая многоядерные архитектуры, параллельные вычисления могут быть организованы как многопроцессным, так и многопоточным выполнением. Многопроцессное выполнение подразумевает оформление каждой подзадачи в виде отдельной программы (процесса). Недостатком такого подхода является сложность организации взаимодействия подзадач. Каждый процесс функционирует в своем виртуальном адресном пространстве, не пересекающемся с адресным пространством другого процесса и для взаимодействия подзадач необходимо использовать специальные средства межпроцессной коммуникации. Многопоточное выполнение подразумевает оформление каждой подзадачи в виде потока в рамках одного процесса. Для взаимодействия потоков не нужно применять какие-либо средства коммуникации. Потоки могут непосредственно обращаться к общим переменным, которые изменяют другие потоки. Но работа с общими переменными приводит к необходимости использования средств синхронизации, регулирующими порядок работы потоков с данными.

Потоки являются более простой основой для реализации параллельных программ по сравнению с процессами. Поэтому параллельная работа множества потоков, решающих общую задачу, более эффективна в плане временных затрат, чем параллельная работа множества процессов.

Предлагаемые методические указания помогут студентам ознакомиться с базовыми принципами построения параллельных программ на языке C# для платформы .NET Framework и овладеть основами параллельного программирования на языке C#, используя потоки. Методические указания содержат необходимые теоретические сведения, а также рекомендации к выполнению лабораторных работ.

1. ОРГАНИЗАЦИИ МНОГОПОТОЧНОГО ПРИЛОЖЕНИЯ

1.1. Одно- и многопоточные процессы

Программное обеспечение по своей организации является многоуровневым. Рассматривая его упрощенно, говорят об уровне ядра системы и уровне приложения, которым соответствуют разные режимы исполнения программного кода: режим с максимальными привилегиями и режим с пониженными привилегиями.

В системе одновременно выполняется более одного пользовательского приложения, не считая системных программ. Каждое приложение – это совокупность взаимосвязанных функций. Для своего исполнения каждая функция требует ресурсов памяти и процессорного времени.

Из приложения пользователя могут осуществляться вызовы функций ядра системы (системные вызовы). Поэтому одновременно в состоянии выполнения наряду с функциями разных приложений могут находиться функциии ядра системы. Но на одном процессоре (ядре) может выполняться только одна функция, а каждая функция требует монопольного использования ресурсов процессора.

Если функция временно прекращает свое исполнение (например, вызывает другую функцию), то программное сотояние её исполнения должно быть сохранено для последующего восстановления (при возобновлении исполнения после возврата управления из вызванной функции). Это состояние сохраняется в стеке процессора. Но функция приложения может обращаться к функциям ядра системы, которые могут также предусматривать вложенные вызовы. Как же разным приложениям удается бесконфликтно выполняться в системе? Это возможно благодаря следующей организации выполнения приложения.

Минимальной единицей исполнения программного кода на уровне функций является поток, а на уровне приложений – процесс. Понятие процесса существовало в операционных системах Windows задолго до появления платформы .Net.

Под процессом понимается выполняющаяся программа (приложение). Формально процесс – это абстракция уровня операционной системы, ис-

пользуемая для описания набора вычислительных ресурсов и необходимой памяти, выделяемой выполняющемуся приложению.

Основная идея управления памятью состоит в том, чтобы процессу выделять не реальную оперативную память, а виртуальную, которую уже потом некоторым образом связать с реальной памятью. Программный код, получаемый при трансляции приложения, и необходимые данные размещаются в виртуальной памяти. На одной из виртуальных страниц находится точка входа в приложение – процедура Main, с которой начинается его выполнение. Но процессор компьютера не может выполнять код и использовать данные, находящиеся в виртуальной памяти, они должны находиться в реальной оперативной памяти. Поэтому при создании процесса приложение загружается в оперативную память, то есть виртуальная страница отображается в страницу реальной оперативной памяти. Вследствие того, что память ограничена, а в системе одновременно может исполняться много процессов, операционная система реализует своппинг – одна страница вытесняется из памяти, а другая загружается. Так реализуется стратегия управления памятью.

Для каждого загруженного в память файла *.exe операционная система создает отдельный изолированный процесс, который применяется на протяжении всего времения его исполнения. Такая изоляция выполнения приложений обеспечивает надёжную и стабильную исполняющую среду, поскольку сбой одного процесса не влияет на функционирование других процессов. Более того, данные одного процесса напрямую не доступны другим процессам, за исключением использования API-интерфейса программирования распределённых вычислений подобного Windows Communication Foundation. Таким образом, процесс можно рассматривать как фиксированное и безопасное окружение (среду) для выполнения приложения. Когда опреационная система создает процесс, то выделяет ему ресурсы.

Каждому процессу Windows назначается уникальный идентификатор процесса (process identifier – PID), процесс может независимо загружаться и выгружаться операционной системой (или программно) при необходи-

мости. Наблюдать исполняемые процессы в системе можно во всем известном окне диспетчера задач Windows (открывается нажатием комбинации клавишь <Ctrl+Shift+Esc>) на вкладке «Процессы». На этой вкладке можно просматривать различные сведения о выполняющихся на машине процессах, в том числе их PID, количество потоков и т.п.

Процесс создаётся для выполнения приложения, но сам не выполняет код приложения, следовательно, время процессора непосредственно процессу не выделяется.

В каждом процессе Windows, который соответствует отдельному приложению, создаётся начальный (основной) поток для исполнения функции Main(), которая служит точкой входа приложения, и возможно другие вторичные потоки.

Каждый процесс имеет своё собственное виртуальное адресное пространство и связанный с ним контекст выполнения. Потоки одного процесса разделяют его адресное пространство. Чтобы потоки могли бесконфликтно использовать ресурсы процессора, каждый поток выполняется в своем контексте, который включает состояние регистров процессора и два стека: стек вызовов потока и стек вызовов ядра, а в рамках контекста процесса создаётся локальное хранилище потоков (Thread Local Storage – STL), в котором хранится контекст каждого потока.

Если приложение не предусматривает при своём исполнении формирования вторичных потоков, то имеем однопоточное приложение и однопоточный процесс его исполнения. Однопоточный процесс часто замедленно реагирует на действия пользователя, когда его единственный поток выполняет сложную операцию.

Именно, чтобы ускорить работу приложения API-интерфейс Windows и платформа .Net позволяют главному потоку порождать дополнительные вторичные потоки для выполнения отдельных функций. Если приложение кроме основного потока создаёт другие вторичные потоки, приложение называется многопоточным и при его исполнении имеем многопоточный процесс.

Но многопоточность не всегда даёт выигрыш. Оформленные как отдельные потоки функции могут выполняться одновременно, представляя собой параллельно выполняемые части кода, если для каждого потока выделена своя процессорная единица (ядро процессора). Иначе, даже будучи оформленные как потоки, выполняемые функции выстроятся в очередь на исполнение к единственному процессору и будут разделять его время в соответствии с установленными для этих потоков приоритетами. Таким образом, один процессор будет выполнять по одному потоку за единицу времени (называемому квантом времени). По истечении выделенного кванта времени выполнение текущего потока будет приостанавливаться, с целью предоставить другому потоку возможность выполнять свою функцию. Когда один поток снимается с выполнения, контекст этого потока сохраняется в локальном хранилище потоков, а вместо него восстанавливается контекст потока, которому предоставлено процессорное время. Такая стратегия управления процессорным временем носит название "вытесняющей приоритетной многозадачности" (задача здесь используется как обозначение потока).

1.2. Классы для управления процессами

В пространстве имён *System.Diagnostics* определено несколько типов, которые позволяют программно взаимодействовать с процессами и различными, связанными с диагностикой средствами (журнал событий, счетчики производительности). С точки зрения организации многопоточных приложений нас могут заинтересовать классы, приведенные в табл.1.1.

Класс *Process* позволяет анализировать процессы, выполняемые на заданной машине (локальной или удалённой), и предоставляет члены, которые позволяют программно запускать и завершать процессы, просматривать и изменять уровень приоритета процесса, получать список активных потоков и загруженных модулей внутри конкретного процесса. Некоторые свойства и методы класса *Process* представлены соответственно в табл.1.2 и 1.3.

Для иллюстрации возможностей класса *Process* ниже приводится метод перечисления запущенных в системе процессов, фрагмент результата показан на рис.1.1.

```
// перечисляет все запущенные в системе процессы
public static void ListAllProcesses()
    {
      Process[] runningProcesses=Process.GetProcesses();

      foreach (Process p in runningProcesses)
      {
        string info = string.Format("-> Pid: {0}\tName: {1}",
                                    p.Id, p.ProcessName);
        Console.WriteLine(info);
      }
    }
```

Таблица 1.1 – Избранные классы пространства *System.Diagnostics*

Класс	Описание
Process	Предоставляет доступ к локальным и удаленным процессам и позволяет запускать и останавливать локальные системные процессы
ProcessModule	Представляет файл с расширением *.dll или *.exe, загруженный в определенный процесс
ProcessModule-Collection	Представляет строго типизированную коллекцию объектов ProcessModule
ProcessStartInfo	Задает набор значений, используемых при запуске процесса с помощью *метода Process.Start()*
ProcessThread	Представляет поток процесса операционной системы. Этот класс применяется для диагностики потоков процесса, но не для создания новых потоков
ProcessThread-Collection	Представляет строго типизированную коллекцию объектов *ProcessThread*

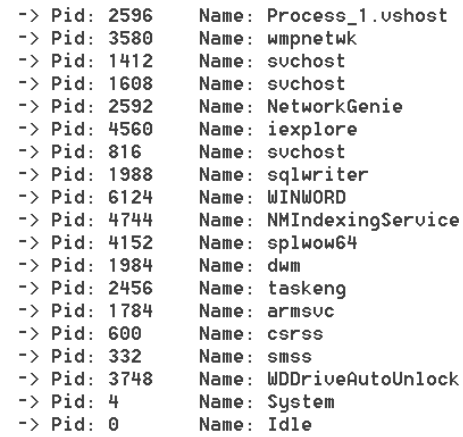

Рис. 1.1. Процессы, выполняемые в системе

Таблица 1.2 – Избранные свойства класса *Process*

Свойство	Описание
ExitTime	*public DateTime ExitTime { get; }*
	Время завершения процесса
Handle	*public IntPtr Handle { get; }*
	Дескриптор назначенный процессу операционной системой
Id	*public int Id { get; }*
	Уникальный идентификатор процесса, созданный операционной системой (PID)
MachineName	*public string MachineName { get; }*
	Имя компьютера, на котором выполняется процесс
MainWindowHandle	*public IntPtr MainWindowHandle { get; }*
	Дескриптор главного окна процесса
MainWindowTitle	*public string MainWindowTitle { get; }*
	Заголовок главного окна процесса
Modules	*public ProcessModuleCollection Modules { get; }*
	Модули, загруженные процессом
ProcessName	*public string ProcessName { get; }*
	Имя процесса
Responding	*public bool Responding { get; }*
	Получает значение, указывающее, отвечает или нет пользовательский интерфейс на ввод
StartTime	*public DateTime StartTime { get; }*
	Время запуска процесса
Threads	*public ProcessThreadCollection Threads { get; }*
	Набор потоков, выполняющихся в процессе

Используя возможности класса *Proces,* можно исследовать как процесс выполнения собственного приложения, так и любые другие процессы на локальном или удалённом компьютере.

Таблица 1.3 – Избранные методы класса *Process*

Метод	Описание
CloseMainWindow()	*public bool CloseMainWindow()*
	Закрывает процесс, имеющий пользовательский интерфейс, посылая сообщение о закрытии главному окну процесса
GetProcessById()	*public static Process GetProcessById(int processId)* *public static Process GetProcessById(int processId, string machineName)*
	Возвращает новый объект *Process* по идентификатору (PID) процесса и имени компьютера в сети. Если второй параметр не задан, то выполняется для локального компьютера
GetProcessesByName()	*public static Process[] GetProcessesByName(string processName)* *public static Process[] GetProcessesByName(string processName,string machineName)*
	Создает массив из новых компонентов *Process* и связывает их со всеми ресурсами процесса на удаленном компьютере с заданным именем. Если второй параметр не задан, то выполняется для локального компьютера
GetCurrentProcess()	*public static Process GetCurrentProcess()*
	Возвращает объект *Process,* представляющий активный процесс в текущий момент
GetProcesses()	*public static Process[] GetProcesses()*
	Возвращает массив объектов процессов, выполняемых на заданной машине
Kill()	*public void Kill()*
	Немедленно останавливает процесс
Start()	*public bool Start()*
	Запускает процесс

2. ОРГАНИЗАЦИЯ ПОТОКА

Среда исполнения .NET CLR предоставляет возможности работы с управляемыми потоками через объект класса Thread пространства имен System.Threading. Thread – это основной класс, который позволяет программно создавать потоки с разными свойствами и управлять их работой.

Каждый поток состоит из следующих составляющих:

- ядро потока;
- блок окружения потока;
- стек пользовательского режима;
- стек режима ядра.

Ядро потока содержит информацию о текущем состоянии потока:

- приоритет потока;
- программный указатель;
- стековые указатели.

Блок окружения потока включает:

- локальное хранилище данных потока;
- заголовок цепочки обработки исключений;
- структуры данных, используемые интерфейсом графических устройств (GDI) и графикой OpenGL.

Стек пользовательского режима используется для организации вложенных вызовов функций потока и передаваемых в методы локальных переменных и аргументов.

Стек режима ядра используется для организации вложенных вызовов функций ядра и передачи аргументов в функции операционной системы в режиме ядра. Ядро ОС вызывает собственные методы и использует стек режима ядра для передачи локальных аргументов, а также для сохранения локальных переменных.

Программный и стековые указатели совместно с регистрами процессора образуют *контекст потока* и позволяют восстановить выполнение потока на процессоре, после его отложенного выполнения.

Создавая приложение, программист не может знать, в каком порядке будут исполняться потоки. Операционная система из набора имеющихся потоков выделяет тот, которому будет передано управление. Для того чтобы поток смог продолжить своё исполнение, сначала должен быть восстановлен контекст этого потока. Если этот выбранный поток принадлежит другому процессу, операционная система переключает для процессора виртуальное адресное пространство, т.е. восстанавливает адресное пространство того процесса, которому принадлежит этот поток.

При восстановлении контекста потока значения из выбранной структуры контекста потока загружаются в регистры процессора.

2.1. Создание и исполнение потока

Компетенцией программиста, разрабатывающего параллельное приложение, является определение, сколько и каких потоков необходимо для решения задачи. Программист сам их создает и контролирует их сотояние в программе.

Среда исполнения .NET CLR предоставляет возможность работы с управляемыми потоками через объект класса *Thread* пространства имен System.Threading. Thread основной класс, который позволяет создавать потоки с разными свойствами и управлять их работой. К основным этапам работы с потоками относятся:

- создание и инициализация потока;
- запуск потока на исполнение;
- ожидание завершения потока.

На этапе создания потока создаётся объект класса *Thread,* инициализация которого определяет метод, выполняемый этим потоком.

Запуск потока на исполнение указывает операционной системе на необходимость перейти к исполнению метода, т.е. поставить поток в очередь готовых к исполнению потоков.

Метод, исполняемый в потоке, определяет результат решаемой подзадачи, который будет сформирован только после завершения потока и необходим для решения основной задачи приложения. Но программист не знает ни момента запуска потока на исполнение, ни момента его заверше-

ния. Определение этого момента, а точнее момента, когда можно начать считывание этого результата, и есть цель этапа ожидания завершения потока.

При создании потока необходимо следовать такой последовательности шагов:

1) Создать метод, который будет служить точкой входа для потока.
2) Создать делегат *ThreadStart* (или *ParameterizedThreadStart*), передав в конструктор в качестве аргумента ссылку на метод, созданный на шаге 1.
3) Создать объект класса *Thread,* передав в конструктор в качестве аргумента делегат, созданный на шаге 2.
4) Установить необходимые свойства созданного на шаге 3 объекта потока (например, имя, приоритет и т.п.).
5) Вызвать *Start()* класса *Thread (*или *Start(object parameter))*, что приведёт к запуску потока для метода, на который ссылается делегат, созданный на шаге 2, при первой возможности (определяется операционной системой).

2.2. Делегаты *ThreadStart, ParameterizedThreadStart*

Согласно шагу 2 для указания на метод, который будет выполняться в потоке, можно использовать два разных типа делегата. Делегат типа

public delegate void ThreadStart()

используется, если метод потока не принимает никаких аргументов и не возвращает результата (т.е. выполняет какие-либо действия в фоновом режиме). Для запуска потока с методом, не принамающим параметров, используется метод *Start()* класса *Thread.*

Делегат типа

public delegate void ParameterizedThreadStart(object obj)

используется, если метод потока должен принимать аргументы и\или возвращать результат. В этом случае следует объявить специальный класс (структуру), в котором определить необходимые поля для аргументов и возвращаемых значений. Для запуска потока с методом, принимающим параметры, используется метод *Start(object obj)* класса *Thread*.

2.3. Способы создания и запуска потока на исполнение

С каждым потоком связывается выполняемая в потоке функция, которая решает отдельную определенную подзадачу в рамках общей задачи приложения. Поэтому в качестве основы потока можно использовать или метод класса, или анонимный метод, или лямбда-выражение. Ниже приводятся примеры разных случаев.

2.3.1. Использование метода класса без параметров

```
public partial class Form1 : Form
  {
     public Form1()
     {
        InitializeComponent();

        // основной  поток
        Thread baseThread = Thread.CurrentThread;
        baseThread.Name = "Поток1";
     }

     public class Printer
     {
        // шаг 1. Создать метод потока
        public void PrintNumbers()
        {
           Console.WriteLine("Начал работу поток {0}",
                                Thread.CurrentThread.Name);
           for (int i = 0; i < 10; i++)
           {
              Console.Write("{0} ", i);
              Thread.Sleep(1000);  // задержка в мсек
           }
         Console.WriteLine("\nЗавершил работу поток {0}\n",
```

```
                                    Thread.CurrentThread.Name);
            }
        }

        private void button1_Click(object sender, EventArgs e)
        {
            Console.WriteLine("\nосновной поток {0} начал работу.",
                                        Thread.CurrentThread.Name);
            Printer myPrinter = new Printer();
            // шаг 2: Создать делегат-метод
            ThreadStart method_for_thread =
                  new ThreadStart(myPrinter.PrintNumbers);
            // шаг 3: Создать объект-поток для выполнения метода
            Thread secondThread = new Thread(method_for_thread);

            // шаг 4:  Определить характеристики потока
            secondThread.Name = "Поток2";

            // шаг 5: Запуск потока на выполнение
            secondThread.Start();

            Console.WriteLine("основной поток {0} завершил работу.",
                                        Thread.CurrentThread.Name);

            MessageBox.Show("Приложение выполнило работу!",
                                "Информация", MessageBoxButtons.OK,
                                 MessageBoxIcon.Information);
        }
    }
```

В примере приложения, результат работы которого представлен на рис.2.1 определено два класса *Form1* и *Printer*.

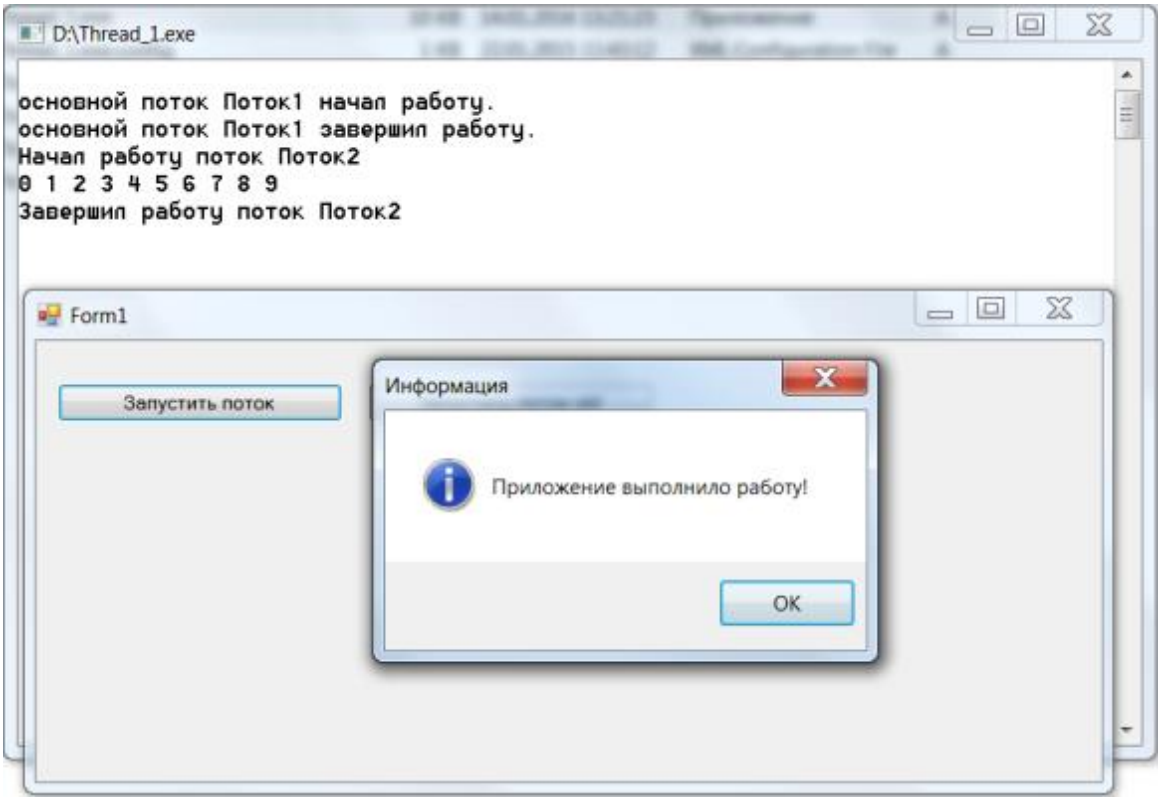

Рис. 2.1. Результат работы приложения

По нажатию кнопки *button1* создается поток, который выполняется параллельно основному потоку приложения. Во вторичном потоке выполняется метод *PrintNumbers()* класса *Printer*, который выводит на консоль с задержкой числа от 0 до 9. Как видно из результата работы приложения, вывод чисел во вторичном потоке выполняется на фоне основного потока, который не ожидает завершения вторичного потока.

2.3.2. Использование анонимного метода без параметров

```
public Form1()
        {
            InitializeComponent();

            // основной  поток
            Thread baseThread = Thread.CurrentThread;
            baseThread.Name = "Поток1";
        }

        private void button2_Click(object sender, EventArgs e)
        {
            Console.WriteLine("\nосновной поток {0} начал работу.",
```

```
                                Thread.CurrentThread.Name);
        // шаг 2,3: Создать объект-поток для выполнения
        //              анонимного метода
        Thread secondThread = new Thread(delegate()
          {
             Console.WriteLine("Начал работу поток {0}",
                                  Thread.CurrentThread.Name);
             for (int i = 0; i < 10; i++)
                {
                   Console.Write("{0} ", i);
                   Thread.Sleep(1000);
                }
             Console.WriteLine("\nЗавершил работу поток {0}\n",
                                  Thread.CurrentThread.Name);
          }
                                                    );
        // шаг 4:  Определить характеристики потока
        secondThread.Name = "Поток2(Анонимный метод)";

        // шаг 5: Запуск потока на выполнение
        secondThread.Start();

        Console.WriteLine("основной поток {0} завершил рабо-
my.", Thread.CurrentThread.Name);
        MessageBox.Show("Приложение выполнило работу!",
                             "Информация", MessageBoxButtons.OK,
                              MessageBoxIcon.Information);
    }
  }
```

Результат работы приложения представлен на рис. 2.2.

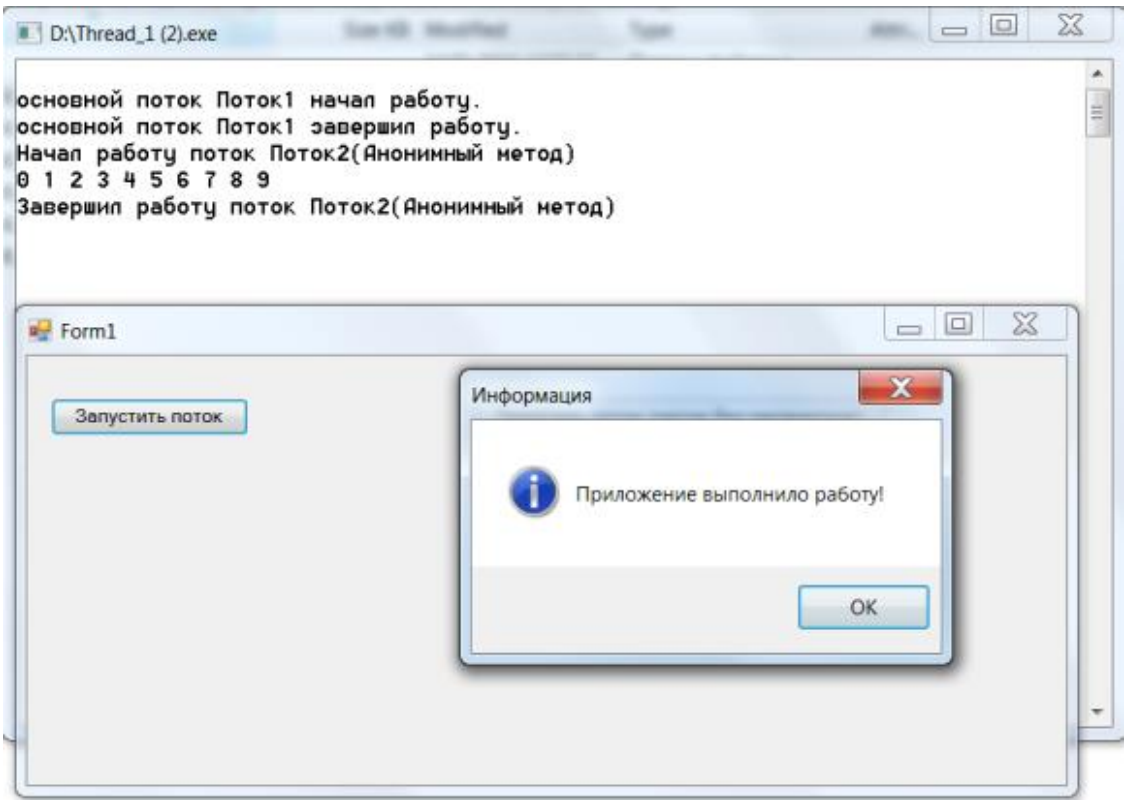

Рис. 2.2. Результат работы приложения

2.3.3. Использование лямбда-выражения без параметров

```
public Form1()
      {
         InitializeComponent();

         // основной  поток
         Thread baseThread = Thread.CurrentThread;
         baseThread.Name = "Поток1";
      }

private void button3_Click(object sender, EventArgs e)
      {
         Console.WriteLine("\nосновной поток {0} начал работу.",
                                 Thread.CurrentThread.Name);
         // шаг 2,3: Создать объект-поток для выполнения
        // Лямбда-выражения
         Thread secondThread = new Thread(() =>
         {
            Console.WriteLine("Начал работу поток {0}",
                                   Thread.CurrentThread.Name);
            for (int i = 0; i < 10; i++)
            {
```

```
            Console.Write("{0} ", i);
            Thread.Sleep(1000);
        }
        Console.WriteLine("\nЗавершил работу поток {0}\n",
                                Thread.CurrentThread.Name);
    }
                                                    );
    // шаг 4:  Определить характеристики потока
    secondThread.Name = "Поток2(Лямбда-выражение)";

    // шаг 5: Запуск потока на выполнение
    secondThread.Start();

    Console.WriteLine("основной поток {0} завершил рабо-
my.", Thread.CurrentThread.Name);
    MessageBox.Show("Приложение выполнило работу!",
                        "Информация", MessageBoxButtons.OK,
                        MessageBoxIcon.Information);
  }
}
```

Результат работы приложения представлен на рис. 2.3.

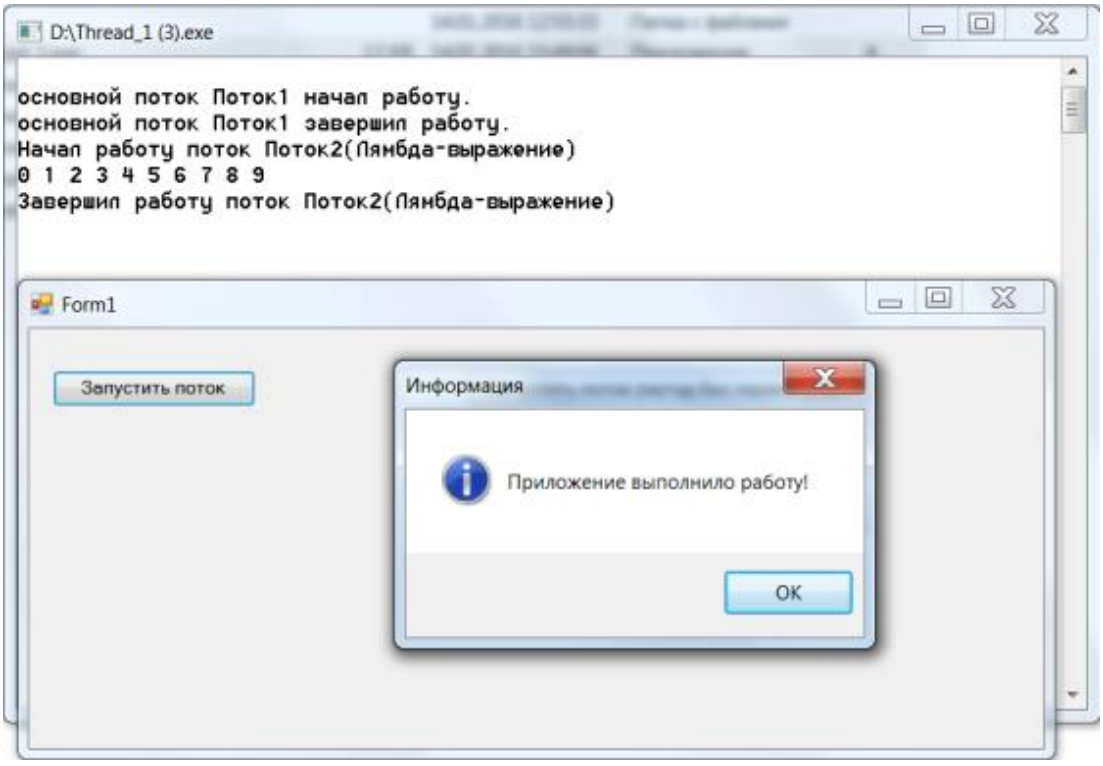

Рис. 2.3. Результат работы приложения

2.3.4. Использование метода класса с параметрами

Для иллюстрации использования метода с параметрами изменим решаемую задачу. Пусть метод (*Sum(object obj)*), выполняемый в потоке, получает массив выводимых на консоль чисел и считает для них сумму.

```
public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();

            // основной  поток
            Thread baseThread = Thread.CurrentThread;
            baseThread.Name = "Поток1";
        }

        public class Params
        {
            private double[] ar;
            private double sum;

            public Params(double[] ar)
            {
                this.ar = ar;
            }

            public double[] Array
            {
                get { return ar; }
            }

            public double Sum
            {
                set { sum = value; }
                get { return sum; }
            }
        }

public class Printer
        {
```

```
        // шаг 1. Создать метод потока
        public void Sum(object obj)
        {
            Console.WriteLine("Начал работу поток {0}",
                                Thread.CurrentThread.Name);

            Params par = (Params)obj;

            for (int i = 0; i < par.Array.Length; i++)
            {
                Console.Write("{0} ", par.Array[i]);
                par.Sum += par.Array[i];
                Thread.Sleep(1000);
            }
            Console.WriteLine("\nЗавершил работу поток {0},
Sum={1}\n", Thread.CurrentThread.Name,par.Sum);
        }
    }

private void button4_Click(object sender, EventArgs e)
    {
        Console.WriteLine("\nосновной поток {0} начал работу.",
                    Thread.CurrentThread.Name);

        Printer myPrinter = new Printer();

        // шаг 2: Создать делегат-метод
        ParameterizedThreadStart method_for_thread =
                    new ParameterizedThreadStart(myPrinter.Sum);

       // шаг 3: Создать объект-поток для выполнения метода
        Thread secondThread = new Thread(method_for_thread);

        // шаг 4:  Определить характеристики потока
        secondThread.Name = "Поток2(метод с параметрами)";

        // шаг 5: Запуск потока на выполнение
        //5.1: подготовка параметров
        double[] src_ar=new  double[10];
        for(int i=0; i<src_ar.Length;i++)
            src_ar[i]=i;
```

```
            Params pars=new Params(src_ar);
            // 5.2: запуск
            secondThread.Start(pars);

            // 5.3:  ожидание результата
            secondThread.Join();

            Console.WriteLine("основной поток {0} завершил работу,
Sum={1}.", Thread.CurrentThread.Name, pars.Sum);

            MessageBox.Show("Приложение выполнило работу!",
                                "Информация", MessageBoxBut-
tons.OK,
                                 MessageBoxIcon.Information);
        }
}
```

Существенных изменений в методе (*button4_Click( )*), запускающем второй поток, не произошло:

- вместо типа делегата *ThreadStart* использовался тип делегата *ParameterizedThreadStart;*
- добавились действия по подготовке аргументов для метода потока;
- добавились действия ожидания завершения потока и вывода результата.

Результат выполнения приложения представлен на рис 3.4. Как здесь видно, из-за ожидания завершения потока, что реализуется вызовом объектного метода *Join()* класса *Thread*, завершение метода *button4_Click( )* происходит только после того, как второй поток завершит свою работу.

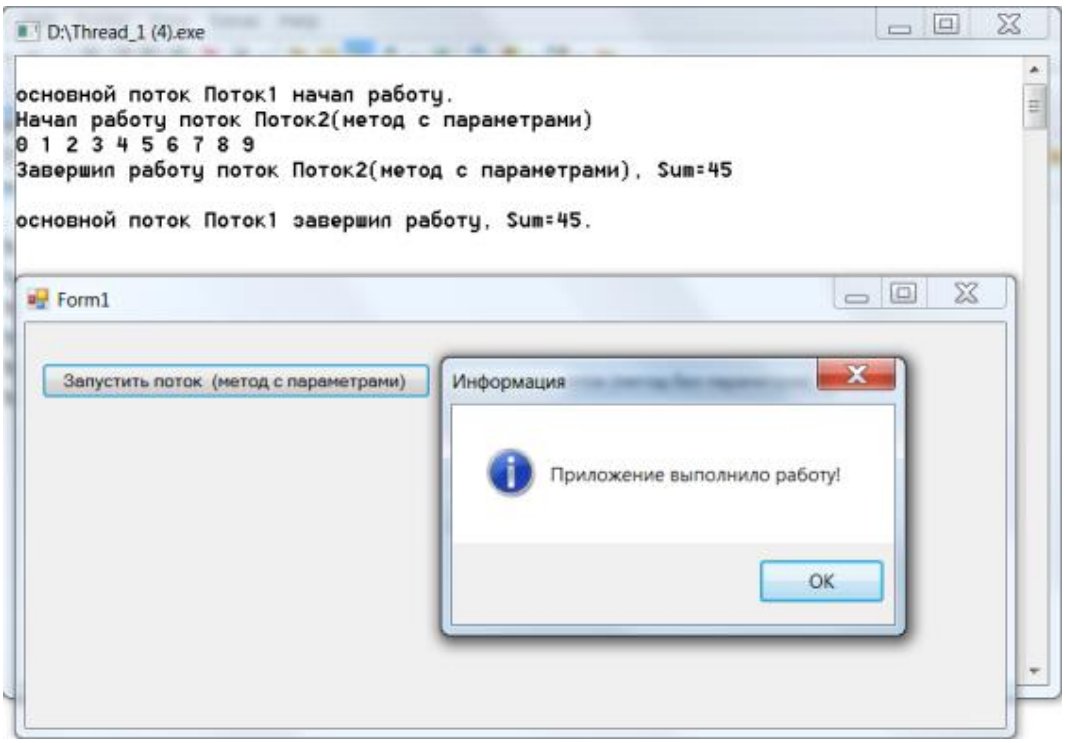

Рис. 3.4. Результат работы приложения

2.3.5. Использование анонимного метода с параметрами

Ниже приводится пример использования анонимного метода с параметрами в качестве метода, образующего поток. Результат выполнения не показан, так как он в точности совпадает с представленным на рис. 1.4.

```
private void button5_Click(object sender, EventArgs e)
      {
          Console.WriteLine("\nосновной поток {0} начал работу.",
                                  Thread.CurrentThread.Name);
         // шаг 2,3: Создать объект-поток для выполнения метода
         Thread secondThread = new Thread(delegate(object obj)
            {
               Console.WriteLine("Начал работу поток {0}",
                                       Thread.CurrentThread.Name);
               Params par = (Params)obj;

               for (int i = 0; i < par.Array.Length; i++)
               {
                  Console.Write("{0} ", par.Array[i]);
                  par.Sum += par.Array[i];
                  Thread.Sleep(1000
```

```
                }
                Console.WriteLine("\nЗавершил работу поток {0},
Sum={1}\n", Thread.CurrentThread.Name, par.Sum);
            }
                                                    );
        // шаг 4:  Определить характеристики потока
        secondThread.Name = "Поток2(анонимный метод с па-
раметрами)";

        // шаг 5: Запуск потока на выполнение
        //5.1: подготовка параметров
        double[] src_ar = new double[10];
        for (int i = 0; i < src_ar.Length; i++)
            src_ar[i] = i;

        Params pars = new Params(src_ar);
        // 5.2: запуск
        secondThread.Start(pars);

        // 5.3:  ожидание результата
        secondThread.Join();

        Console.WriteLine("основной поток {0} завершил работу,
Sum={1}.", Thread.CurrentThread.Name, pars.Sum);
        MessageBox.Show("Приложение выполнило работу!",
                        "Информация",
                         MessageBoxButtons.OK,
                         MessageBoxIcon.Information);
    }
```

2.3.6. Использование лямбда-выражения с параметрами

Ниже приводится пример использования лямбда-выражения с параметрами в качестве метода, образующего поток. Результат выполнения не приводится, так как он в точности совпадает с представленным на рис. 1.4.

```
private void button6_Click(object sender, EventArgs e)
    {
        Console.WriteLine("\nосновной поток {0} начал работу.",
                          Thread.CurrentThread.Name);
        // шаг 2,3: Создать объект-поток для выполнения ме-
тода
```

```
        Thread secondThread = new Thread((object obj)=>
        {
            Console.WriteLine("Начал работу поток {0}",
                                Thread.CurrentThread.Name);
            Params par = (Params)obj;

            for (int i = 0; i < par.Array.Length; i++)
            {
                Console.Write("{0} ", par.Array[i]);
                par.Sum += par.Array[i];
                Thread.Sleep(1000);
            }
            Console.WriteLine("\nЗавершил работу поток {0},
Sum={1}\n", Thread.CurrentThread.Name, par.Sum);
        }
                                                    );
        // шаг 4:  Определить характеристики потока
        secondThread.Name = "Поток2(лямбда выражение с па-
раметрами)";

        // шаг 5: Запуск потока на выполнение
        //5.1: подготовка параметров
        double[] src_ar = new double[10];
        for (int i = 0; i < src_ar.Length; i++)
            src_ar[i] = i;

        Params pars = new Params(src_ar);
        // 5.2: запуск
        secondThread.Start(pars);

        // 5.3:  ожидание результата
        secondThread.Join();

        Console.WriteLine("основной поток {0} завершил работу,
Sum={1}.", Thread.CurrentThread.Name, pars.Sum);

        MessageBox.Show("Приложение выполнило работу!",
                                "Информация",
                                 MessageBoxButtons.OK,
                                 MessageBoxIcon.Information);
    }
```

2.3.7. Передача аргументов через глобальные переменные

В примерах с анонимным методом и лямбда-выражением (см. пп. 2.4.5-2.4.6) для запуска потока на исполнение использовался метод *Start(object args)* класса *Thread*. Но используя механизм анонимных методов, можно было использовать метод *Start()* класса *Thread*, т.е. обойтись без явной передачи аргументов, а необходимые данные передавать через глобальные переменные. Как показано в примере, в роли такой переменной выступает объект *par* класса *Params*:

```
private void button8_Click(object sender, EventArgs e)
      {
         Console.WriteLine("\nосновной поток {0} начал работу.",
                                 Thread.CurrentThread.Name);

         // подготовка параметров
         double[] src_ar = new double[10];
         for (int i = 0; i < src_ar.Length; i++)
            src_ar[i] = i;

         Params par = new Params(src_ar);

         // шаг 2,3: Создать объект-поток для выполнения метода
         Thread secondThread = new Thread(() =>
         {
            Console.WriteLine("Начал работу поток {0}",
                                    Thread.CurrentThread.Name);
            for (int i = 0; i < par.Array.Length; i++)
            {
               Console.Write("{0} ", par.Array[i]);
               par.Sum += par.Array[i];
               // задерживает выполнение тек потока на 1сек
               Thread.Sleep(1000);
            }
            Console.WriteLine("\nЗавершил работу поток {0},
Sum={1}\n", Thread.CurrentThread.Name, par.Sum);
         }
                                                                  );
         // шаг 4:  Определить характеристики потока
```

```
        secondThread.Name = "Поток2(лямбда-выражение с глобальными параметрами)";

        // шаг 5: Запуск потока на выполнение
        secondThread.Start();

        // 5.3:  ожидание результата
        secondThread.Join();

        Console.WriteLine("основной поток {0} завершил работу, Sum={1}.", Thread.CurrentThread.Name, par.Sum);

        MessageBox.Show("Приложение выполнило работу!",
                        "Информация", MessageBoxButtons.OK,
                        MessageBoxIcon.Information);
    }
```

2.3.8. Передача аргументов через объект массив

Тип делегата *ParameterizedThreadStart* предполагает передачу в метод потока только одного параметра типа *object*. Для передачи большего количества аргументов и возвращаемых из метода значений приходится создавать специальный класс. В приведенных выше примерах это был класс *Params.*

Оказывается, этого можно не делать, если все параметры представить как элементы массива *object[ ],* как показано в следующем примере.

```
public void Calculator(object obj)
    {
        Console.WriteLine("Начал работу поток {0}",
                          Thread.CurrentThread.Name);

        object[] pars = (object[])obj;
        double par1 = (double)pars[0];
        string oper = (string)pars[1];
        double par2 = (double)pars[2];
        double res=0;
```

```
            int i=0;
            foreach (object el in pars)
               Console.WriteLine("pars[{0}]={1}", i, pars[i++]);
            switch (oper)
            {
               case "+": res = par1 + par2;
                  break;
               case "-": res = par1 - par2;
                  break;
               case "*": res = par1 * par2;
                  break;
               case "/": res = par1 / par2;
                  break;
            }
            pars[3] = (object)res;

            Console.WriteLine("\nЗавершил работу поток {0}, res={1}\n", Thread.CurrentThread.Name, pars[3]);
         }

         private void button9_Click(object sender, EventArgs e)
         {
            Console.WriteLine("\nосновной поток {0} начал работу.", Thread.CurrentThread.Name);
            // шаг 2: Создать делегат-метод для потока
            ParameterizedThreadStart method_for_thread =
                        new ParameterizedThreadStart(Calculator);
            // шаг 3: Создать объект-поток для выполнения метода
            Thread secondThread = new Thread(method_for_thread);

            // шаг 4:  Определить характеристики потока
            secondThread.Name = "Поток2(передача параметров через массив)";
```

```
            // шаг 5.1: подготовка параметров
            object[] args = new object[4];
            args[0] = 125.99;
            args[1] = "+";
            args[2] = 25.01;
            args[3] = null;

            // шаг 5.2: Запуск
            secondThread.Start(args);

            // 5.3:  ожидание результата
            secondThread.Join();

            Console.WriteLine("основной поток {0} завершил работу,
{1} {2} {3} ={4}.", Thread.CurrentThread.Name, args[0], args[1], args[2],
args[3]);

            MessageBox.Show("Приложение выполнило работу!",
                                    "Информация", MessageBoxButtons.OK,
                                MessageBoxIcon.Information);
        }
```

Действенность рассмотренного способа передачи и возврата значений в поток представлена на рис. 2.5.

Может быть, в этом случае придётся написать немного больше кода, но не потребуется создавать класс для передачи параметров в поток.

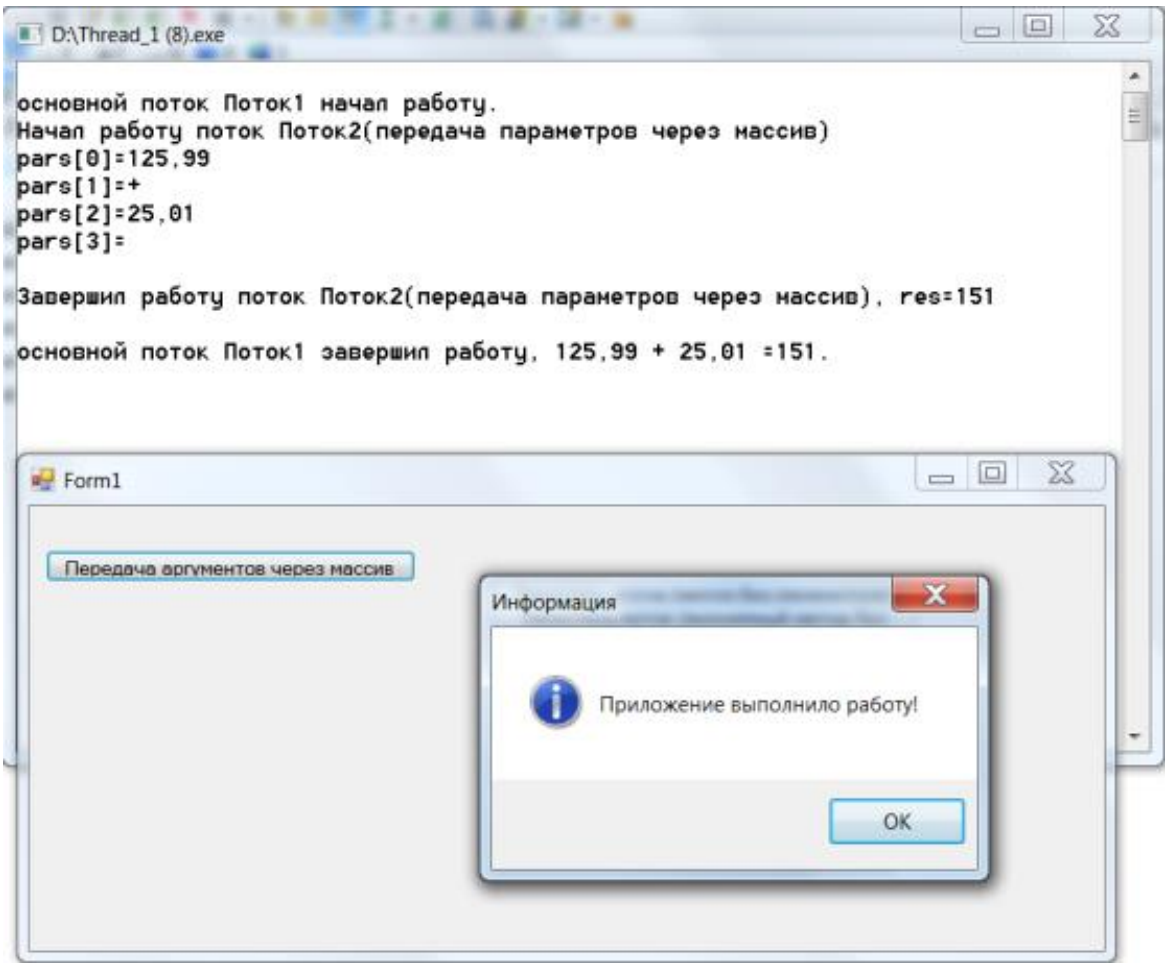

Рис. 2.5. Результат работы приложения

3. ФУНКЦИОНАЛЬНЫЕ ВОЗМОЖНОСТИ КЛАССА THREAD

Функциональные возможности класса *Thread* определяются его свойствами и методами.

3.1. Свойства

Некоторые свойства класса *Thread* представлены в табл.3.1.

Таблица 3.1 – Свойства класса *Thread*

Свойство	Описание
Name	*public string Name {get;set;}*
	Имя потока
ManagedThreadId	*public int ManagedThreadId { get; }*
	Уникальный идентификатор потока
Priority	*public ThreadPriority Priority {get; set;}*
	Планируемый приоритет потока
IsBackground	*public bool IsBackground{get; set;}*
	Статус фонового потока
IsAlive	*public bool IsAlive { get; }*
	Статус выполнения текущего потока
IsThreadPoolThread	*public bool IsThreadPoolThread { get; }*
	Статус принадлежности к группе управляемых потоков
ThreadState	*public ThreadState ThreadState { get; }*
	Состояния текущего потока
CurrentThread	*public static Thread CurrentThread { get; }*
	Объект выполнящегося потока
CurrentContext	*public static Context CurrentContext {get;}*
	Текущий контекст, в котором выполняется поток.
CurrentCulture	*public CultureInfo CurrentCulture {get;set;}*
	Язык и региональные параметры текущего потока

3.1.1.Имя потока

Свойство Name позволяет задавать потокам собственные имена, отличные от служебных имён, получаемых в момент создания. Именование потоков облегчает процесс отладки многопоточного приложения.

3.1.2.Уникальный идентификатор потока

Каждому потоку при создании присваивается уникальный идентификатор потока, который можно узнать, используя объектное свойство *ManagedThreadId.*

3.1.3.Назначение приоритета потока

Приоритет потока используется для определения порядка выполнения потоков при распределении процессорного времени. Из двух потоков, готовых к выполнению, на выполнение будет выбран тот, у кого больше приоритет.

Если в процессе выполнения потока появился готовый к выполнению поток с большим приоритетом, то выполнение текущего потока будет приостановлено, даже если не истек отведенный ему квант времени.

Все потоки распределяются по группам приоритетности, потоки из одной группы могут быть выбраны на выполнение только в том случае, если нет готовых к выполнению потоков в группах с высшей приоритетностью.

Если некоторое приложение, имеющее низкий приоритет, долго не выполнялось, то ОС временно повышает его приоритет, так что и оно начнет выполняться.

Все потоки в группе с одинаковым приоритетом выстраиваются в очередь. Каждому из них в соответствии с очередью отводится на выполнение некоторый квант времени процессора. По истечении этого кванта поток снимается с выполнения, и в состояние выполнения переводится следующий по очереди поток.

Свойство *Priority* позволяет назначить потоку приоритет. Существует пять градаций приоритета потока, определяемых типом *ThreadPriority,* значения которого представлены в табл.3.2. Приоритеты потоков опреде-

ляют очередность выделения и время доступа к ЦП. Высокоприоритетные потоки имеют преимущество и чаще получают доступ к ЦП, чем низкоприоритетные.

Таблица 3.2 – Приоритеты потоков

Метод	Описание
AboveNormal	Задаёт приоритет не выше ***Highest*** и не ниже ***Normal***
BelowNormal	Задаёт приоритет не выше ***Normal*** и не ниже ***Lowest***
Highest	Задаёт наивысший приоритет
Lowest	Задаёт наинизший приоритет
Normal	Задаёт приоритет не выше ***AboveNormal*** и не ниже ***BelowNormal***

Уровень приоритета можно поменять в любой момент. Но приоритет потока не влияет на состояние потока. Чтобы операционная система могла планировать выполнение потока, состояние потока должно иметь значение *Running.*

По умолчанию поток имеет ***Normal*** (средний) приоритет.

Необходимо понимать, что приоритет потока сказывается только в случае конкуренции множества потоков за мощности ЦП.

Значение, определяемое свойством *Priority,* является рекомендательным, но реальный приоритет устанавливает планировщик операционной системы. Приоритеты используются планировщиком так, что потоки с одинаковым уровнем приоритета будут получать одинаковые кванты времени для своей работы.

Планирование выполнения потоков осуществляется с учетом их приоритета, но алгоритмы планирования, которые используются для определения порядка выполнения потоков, в разных операционных системах отличаются. Операционная система может изменять приоритеты потоков

динамически при перемещении фокуса пользовательского интерфейса между приложениями переднего плана и фоновыми приложениями.

Менять приоритет ***Normal*** на другой надо аккуратно, иначе можно получить негативный эффект.

Установка приоритета потока на ***Highest*** не означает работу потока в реальном времени, так как существует еще приоритет процесса приложения. Для того чтобы поток работал в режиме реального времени, используется класс *Process* из пространства имен *System.Diagnostics* для поднятия приоритета:

Process.GetCurrentProcess().PriorityClass = ProcessPriorityClass.High;

От значения приоритета *ProcessPriorityClass.High* один шаг до наивысшего приоритета процесса – *Realtime*. И если процесс в режиме приоритета *Realtime* случайно попадёт в бесконечный цикл, то операционная система может быть полностью заблокирована. Единственным спасением в этом случае может быть только выключение питания или перезагрузка компьютера. По этой причине ProcessPriorityClass.High считается максимальным приоритетом процесса, рекомендованным к использованию пользователям.

3.1.4.Фоновые потоки

Потоки в системе делятся на потоки основные и фоновые.

Тип потока определяет установленное для него свойство *IsBackground* и влияет способ завершения процесса.

CLR завершает процесс только в том случае, если все основные потоки, принадлежащие процессу, завершились.

Если при необходимости завершить процесс он имеет незавершенные фоновые потоки, то они останавливаются и не завершаются.

3.1.5.Состояние выполнения потока

Проверить, находится поток в состоянии выполнения, можно, используя объектное свойство *IsAlive.* Это свойство имеет значение *true,* если поток не перешел в состояние завершения выполнения.

3.1.6.Принадлежность потока к группе управляемых потоков

Программист при необходимости выполнить какие-либо подзадачи приложения в параллельном режиме может не утруждать себя действиями, связанными с запуском и управлением потоков. Для этого он может воспользоваться классом *ThreadPool,* который предоставляет возможности выполнить метод класса, поставив его в очередь пула потоков. Если поток выполняется через пул, то его свойство *IsThreadPoolThread* имеет значение *true,* т.е. поток принадлежит к группе управляемых потоков. Если поток выполняется без постановки его в очередь пула, то это свойство имеет значение *false.*

3.1.7.Состояния потока

Свойство *ThreadState* предоставляет более подробную информацию о состоянии потока, чем свойство *IsAlive.* Каждый поток может находиться в одном из нескольких состояний, описанных в табл.3.3. Изменение состяония потока происходит при выполнении определённых действий над потоком, которые вызываются методами управления потоками класса *Thread.*

Упрощенно диаграмма состояний потока от момента его создания до завершения выполнения представлена на рис. 3.1.

После создания потока и необходимой инициализации поток переходит в состояние «готов», занимая в своей группе приоритетности место в конце очереди. Состояние «готов» означает, что поток готов к выполнению и ожидает предоставления доступа к процессору. Поток, который выполняется в текущий момент времени, имеет статус «выполнение». Каждый поток во время своего выполнения многократно может прерывать своё выполнение и событиями, не связанными с выделением процессорного времени. События, приводящие к приостановке выполнения потока (состояние «ожидание»), могут быть асинхронными (внешними) или синхронными (вызванные самим потоком). При выполнении операций ввода-вывода или вызове функций ядра поток всегда переводится в состояние «ожидание».

Таблица 3.3 – Состояния потока

Состояние	Из состояния	Описание состояния
Unstarted		Начальное состояние. Объект потока создан. Наступает в момент создания объекта потока
Running	***Unstarted Abort-Requested WaitSleep-Join***	Поток в состоянии выполнения. Наступает в результате выполнения метода *Start()* над объектом потока и означает, что поток принят операционной системой на выполнение. Так же может наступать при восстановлении работы потока после *AbortRequested* или *WaitSleepJoin*
Background		Поток выполняется как фоновый поток, в противоположность потокам переднего плана. Это состояние управляется заданием свойства *IsBackground*
WaitSleep-Join	***Running***	Поток в состоянии блокировки выполнения. Это может произойти в результате вызова методов: *Sleep()* из метода этого потока; *Join()* для этого метода из вызывающего потока; при вызове метода *Monitor.Enter()* или *Monitor.Wait()* из метода этого потока; в результате ожидания объекта синхронизации потока, такого как *ManualResetEvent*
Abort-Requested	***WaitSleep-Join Running***	Поток в состоянии обработки требования завершения потока (вызван метод *Abort()*) – поток получил исключение *ThreadAbortException* и обрабатывает его
Aborted	***Abort-Requested***	Переходит в это состояние, если при обработке исключения *ThreadAbortException* завершение потока не отменено методом *ResetAbort()*. Поток уже не выполняет работу, но его состояние еще не изменилось на Stopped
Stopped	***Running Aborted***	Поток был остановлен. Это может произойти в результате: нормального завершения метода потока, обработки иключений: *ThreadInterruptedException*, *ThreadAbortException*
Suspend-Requested	***Running***	Запрашивается приостановка работы потока вызовом метода *Suspend()* из другого потока
Suspended	***Suspend-Requested***	Поток был приостановлен

После завершения таких операций или возврата управления из функций поток переводится в состояние «готов» и помещается в очередь готовых потоков. В состояние «остановлен» поток переводится, когда метод потока завершил исполнение или поток будет снят с исполнения.

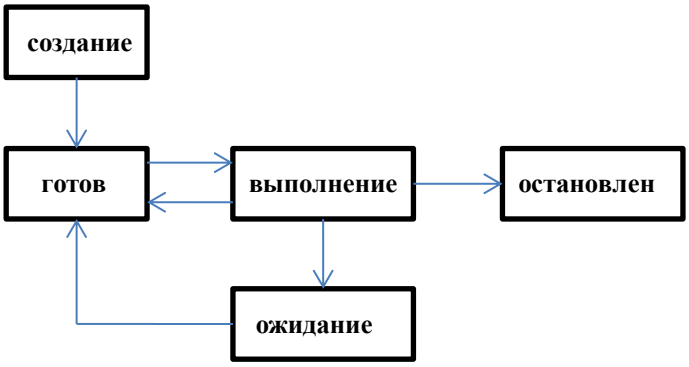

Рис. 3.1. Упрощенная диаграмма состояний выполнения потока

Надо помнить, понимать и учитывать при работе с потоками, что при переводе потока в состояние «готов» или выведении из него могут выполняться действия, связанные с загрузкой (выгрузкой) контекста.

Если выбранный для исполнения поток принадлежит другому процессу, Windows переключает для процессора виртуальное адресное пространство, после чего загружается контекст потока – значения из выбранной структуры контекста потока загружаются в регистры процессора .

Методы класса *Thread* для управления потоками

Некоторые методы класса *Thread* для управления потоками представлены в табл.3.4.

3.2.1. Домены процесса

Домены процесса (Application Domain – класс *System.AppDomain*) – это логические подразделы внутри отдельного процесса, в которых размещается набор связанных сборок .Net.

Таблица 3.4 – Методы класса *Thread*

Метод	Описание
GetDomainID()	*public static int GetDomainID()*
	Возвращает уникальный идентификатор домена приложения.
GetDomain()	*public static AppDomain GetDomain()*
	Возвращает домен, в котором выполняется текущий поток.
Start()	*public void Start()* *public void Start(object parameter)*
	Планирует выполнение потока, вынуждая ОС систему изменить его состояние ThreadState.Running, передавая объект с данными в качестве аргумента для метода потока.
Join()	*public void Join()* *public bool Join(int millisecondsTimeout)* *public bool Join(TimeSpan timeout)*
	Блокирует вызывающий поток до завершения потока, для которого вызывается этот метод или до истечения заданного времени. Возвращает true, если поток завершился
Sleep()	*public static void Sleep(int millisecondsTimeout)* *public static void Sleep(TimeSpan timeout)*
	Приостанавливает текущий поток на заданное время
Interrupt()	*public void Interrupt()*
	Прерывает состояние блокировки потока *WaitSleepJoin*, с возможностью его завершения или продолжения исполнения
Abort()	*public void Abort()*
	Вызывает исключение ThreadAbortException в вызвавшем его потоке для того, чтобы начать процесс завершения потока, для которого вызван этот метод. Обычно завершает поток
ResetAbort()	*public static void ResetAbort()*
	Отменяет метод Abort, запрошенный для текущего потока
Suspend()	*public void Suspend()*
	Приостанавливает работу потока (устарел)
Resume()	*public void Resume()*
	Возобновляет приостановленную работу потока (устарел)
SpinWait()	*public static void SpinWait(int iterations)*
	Вынуждает поток выполнять ожидание столько раз, сколько определено параметром *iterations*
Yield()	*public static bool Yield()*
	Позволяет вызвавшему потоку передать выполнение другому потоку, который выбирает ОС, готовому к использованию

Для получения домена, в котором выполняется текущий поток, неохо-димо вызывать метод *GetDomain()*.

Домены в .Net используются как механизм, позволяющий запустить группу приложений в одном процессе, обеспечивая относительную изоляцию приложений друг от друга, в то же время, позволяют им взаимодействовать друг с другом значительно быстрее, чем в случае отдельных процессов.

Уникальный идентификатор домена приложения можно получить, ис-пользуя метод *GetDomainID()*.

3.2.2. Метод Start()

Запуск потока на исполнение происходит вызовом метода Start() с возможностью задать аргументы для метода потока. Между запуском метода *Start()* и началом выполнения метода потока может пройти некоторое время. Метод потока не начнёт выполняться немедленно. Метод Start() сообщает операционной системе, что поток должен начать своё исполнение с заданным приоритетом и приводит к изме-нению состояния потока с *Unstarted* на *Running*, которое произойдет, как только планировщик потоков операционной системы выберет по-ток для исполнения. До этого момента поток будет сохранять состоя-ние *Unstarted*, а метод *Start()* не закончит своё исполнение. Таким можно, говорить, что метод *Start()* запускает поток на исполнение, но когда это произойдёт, решает операционная система.

3.2.3. Метод Join()

Используется для блокировки вызывающего этот метод потока до завершения потока, для которого применяется *Join()* или до истече-ния заданного как аргумента времени.

Если при запуске метода указан аргумент время блокировки, то возвращается значение true, если поток завершился, иначе *false*.

Если вызов *Join()* выполняется без аргумента, то вызывающий этот метод поток будет заблокирован до момента завершения потока, для которого был вызван метод *Join()*.

3.2.4. Метод Sleep()

С помощью метода *Sleep()* выполнение потока можно приостановить на заданное время (количество миллисекунд), поток преводится в состояние *WaitSleepJoin*, и по истечении заданного времени должен возбновить свою работу.

Если время, на которое приостанавливается поток, равно нулю, поток освобождает оставшуюся часть своего интервала времени для любого потока с таким же приоритетом, готовым к выполнению. Если других, готовых к выполнению потоков с таким же приоритетом нет, выполнение текущего потока не приостанавливается.

Использование этого метода допустимо, но на практике для синхронизации потоков рекомендуется использовать другие средства. Главной причиной этого является следующее.

Если поток должен «уснуть» на заранее известное время, можно вызвать метод *Sleep()*, указав ему это время достаточно точно. Но будет ли поток спать точно столько, сколько указано при вызове *Sleep()?* В общем случае, а, может, и наверняка,– нет. Это связано с тем, что по истечении указанного периода времени поток не получает немедленного доступа к процессору. В этот момент может выполняться другой, более высокоприоритетный поток, и пробуждающемуся потоку придётся подождать своей очереди.

Для решения проблемы предусмотрены специальные значения констант: *Timeout.Infinite* и *Timeout.InfiniteTimeSpan.* Если указать эти константы как аргумент *Sleep()*, то поток заснёт навсегда. Для того чтобы пробудить поток в необходимый момент времени, используются методы *Interrupt()* и *Abort().*

3.2.5. Метод Interrupt()

Вызывающий поток использует метод *Interrupt()* для вывода потока из состояния блокировки. Метод *Interrupt()* не оказывает никакого воздействия на прерываемый поток, если поток не находится в заблокированном состоянии *WaitSleepJoin.*

Если прерываемый поток находится в состоянии *WaitSleepJoin*, то при выполнении метода *Interrupt()* выбрасывается исключение *ThreadInterruptedException* и *у*правление передается обработчику этой исключительной ситуации, который определит дальнейшее состояние прерываемого потока.

Реакция на исключение *ThreadInterruptedException* должна быть предусмотрена в методе, определяющем прерываемый поток. Если эта обработка не предусмотрена, то сработает системный обработчик исключительной ситуации и работа приложения будет завершена аварийно и немедленно, что является ненормальным для работы приложения.

Поэтому использование метода *Interrupt()* предполагает наличие обработчика исключения *ThreadInterruptedException*, который должен размещаться в операторе *try-catch* в методе, организующем прерываемый поток.

Иллюстрацию описанного механизма рассмотрим на примере, в котором класс *Counter* содержит метод *Count()* для организации потока, запускаемого на выполнение в методе *button1_Click()* класса главной формы приложения Form1, результат работы которого представлен на рис.3.2.

```
class Counter
{
   public int counter = 0;

   public void Count()
   {
      try
      {
          while (counter<100000)
          {
              Console.WriteLine("Count state: {0}",
                                  Thread.CurrentThread.ThreadState);
```

```
            Console.WriteLine("{0}", counter++);
            Thread.Sleep(0); // перевод в состояние ожидания
        }
    }
    catch (ex)
    {
        Console.WriteLine("ThreadInterruptedException: {0}",
                                    ex.Message);
        return;
    }
    finally
    {
        Console.WriteLine("finally");
    }

    Console.WriteLine("Count end!");
  }
}
```

В методе *Count()* увеличивается в цикле счетчик, эти действия помещены в блок *try* и обрабатывается исключение ThreadInterrupted-Exception. Важно отметить на вызов Thread.Sleep(0). Если его убрать, то метод потока прерываться не будет, поскольку исключение выбрасывается только тогда, когда поток находится в заблокированном состоянии.

```
// иллюстрация метода Interrupt()
private void button1_Click(object sender, EventArgs e)
   {
      Counter c = new Counter();
      Thread thr = new Thread(c.Count);
      thr.Start();
      Thread.Sleep(100); // чтоб счетчик успел насчитать до 10
      if (c.counter >= 10)
         thr.Interrupt();
```

```
        thr.Join();
        Console.WriteLine("thr state: {0} after Join()", thr.ThreadState);
    }
```

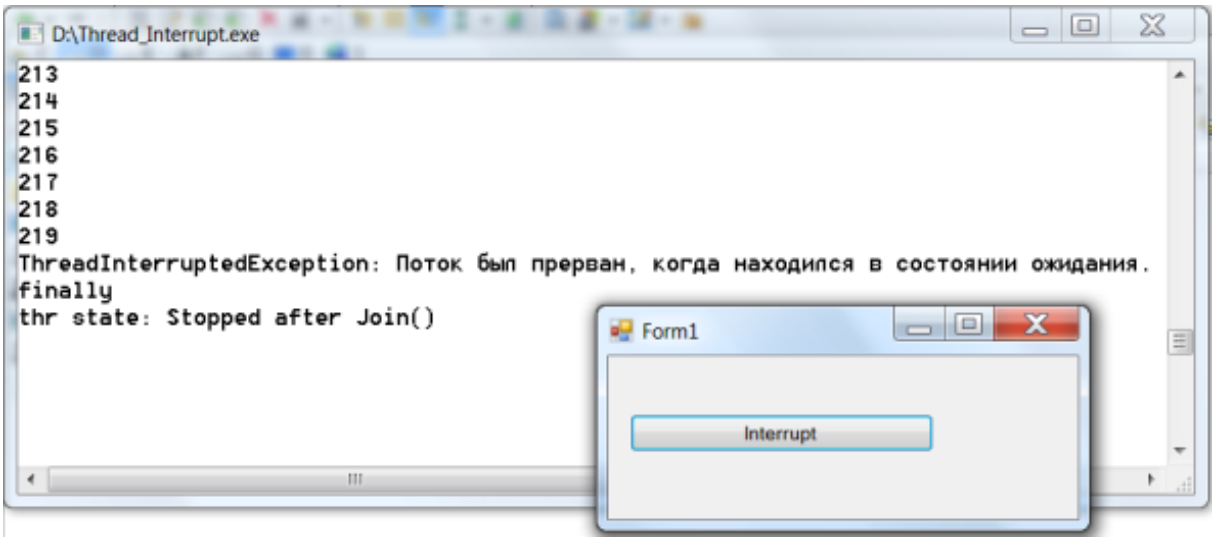

Рис. 3.2. Результат работы приложения

В рассмотренном примере выведенный из состояния блокировки поток с помощью метода *Interrupt()* завершает свою работу. Но если бы обработчик исключения был написан иначе (например, передавал управление на оператор *try)*, то метод потока не завершился бы, пока не отработал бы до конца):

```
    public void Count()
        {
    label:
            try
            {
                while (counter<100000)
                {
                    Console.WriteLine("Count state: {0}",
                                      Thread.CurrentThread.ThreadState);
                    Console.WriteLine("{0}", counter++);
                    Thread.Sleep(0); // перевод в состояние ожидания
                }
            }
```

```
        catch (ex)
        {
            Console.WriteLine("ThreadInterruptedException: {0}",
                                    ex.Message);
            goto label;
        }
        finally
        {
            Console.WriteLine("finally");
        }

        Console.WriteLine("Count end!");
    }
```

3.2.6. Метод Abort()

Метод *Abort()* завершает выполнение потока, для которого он вызывается (как правило, из другого потока), независимо от его состояния.

Аналогично методу *Interrupt(),* метод *Abort()* выбрасывает исключение ThreadAbortException. Обрабатывая это исключение, можно как завершить поток, так и продолжить его работу, отменив завершение.

Поскольку метод *Abort()* не всегда способен остановить поток мгновенно, и если требуется завершить поток раньше, чем продолжить выполнение программы, то после метода *Abort()* следует сразу же вызвать метод *Join()*. Пример использования метод *Abort()* для завершения потока:

```
class Counter
    {
        public int counter = 0;

        public void Count()
        {
```

```
            try
            {
                while (counter < 50)
                {
                    Console.WriteLine("{0}", counter++);
                }
            }
            catch (ThreadAbortException ex)
            {
                Console.WriteLine("ThreadAbortException:                {0}",
ex.Message);
                return;
            }
            finally
            {
                Console.WriteLine("finally");
            }
            Console.WriteLine("Count end!");
        }
    }
    // иллюстрация метода Abort()
    private void button2_Click(object sender, EventArgs e)
    {
        Counter c = new Counter();

        Thread thr = new Thread(c.Count);
        thr.Start();

        Thread.Sleep(10);   // чтоб немного поработал

        thr.Abort();

        thr.Join();
        Console.WriteLine("thr state: {0} after Join()", thr.ThreadState);
```

```
}
```

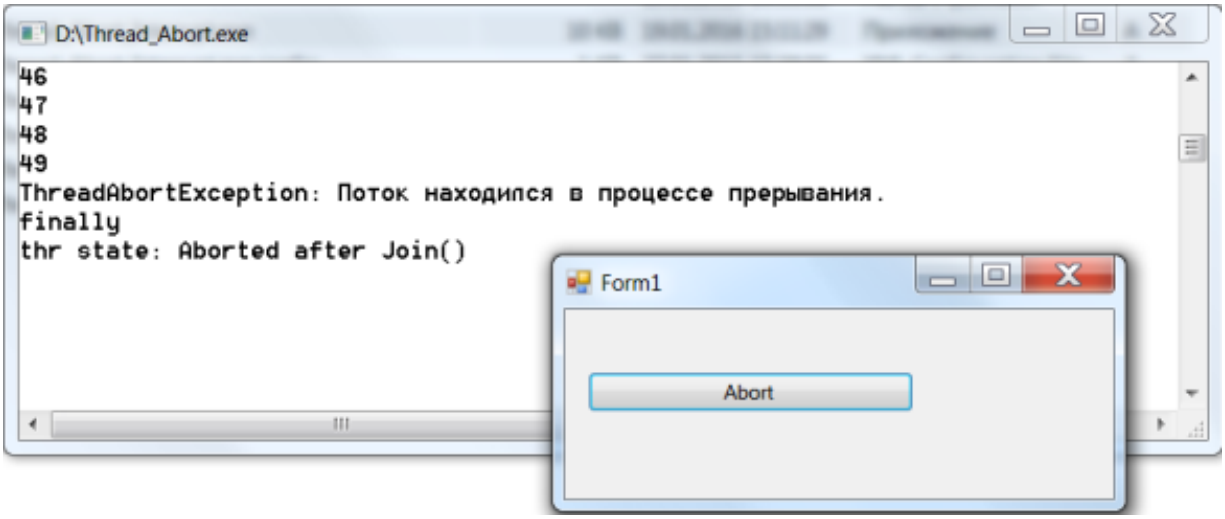

Рис. 3.3. Результат работы приложения

3.2.7. Метод ResetAbort()

Этот метод позволяет отменить завершение потока, вызванное методом *Abort()*. Его необходимо вызвать в блоке catch при обработке исключения *ThreadAbortException.* Но простой его вызов приведет к непосредственно к завершению потока. Чтобы реанимировать работу потока, необходимо передать управление какому-либо оператору в методе потока, например, как показано далее:

```
public void Count__()
        {
label:
            try
            {
                while (counter < 50)
                {
                    Console.WriteLine("{0}", counter++);
                }
            }
            catch (ThreadAbortException ex)
            {
                Console.WriteLine("ThreadAbortException: {0}", ex.Message);
```

```
            Thread.ResetAbort();
            goto label;
        }
        finally
        {
            Console.WriteLine("finally: Возобновляем работу потока!");
        }
        Console.WriteLine("Count end!");
    }
```

Результат работы приложения представлен на рис.3.4.

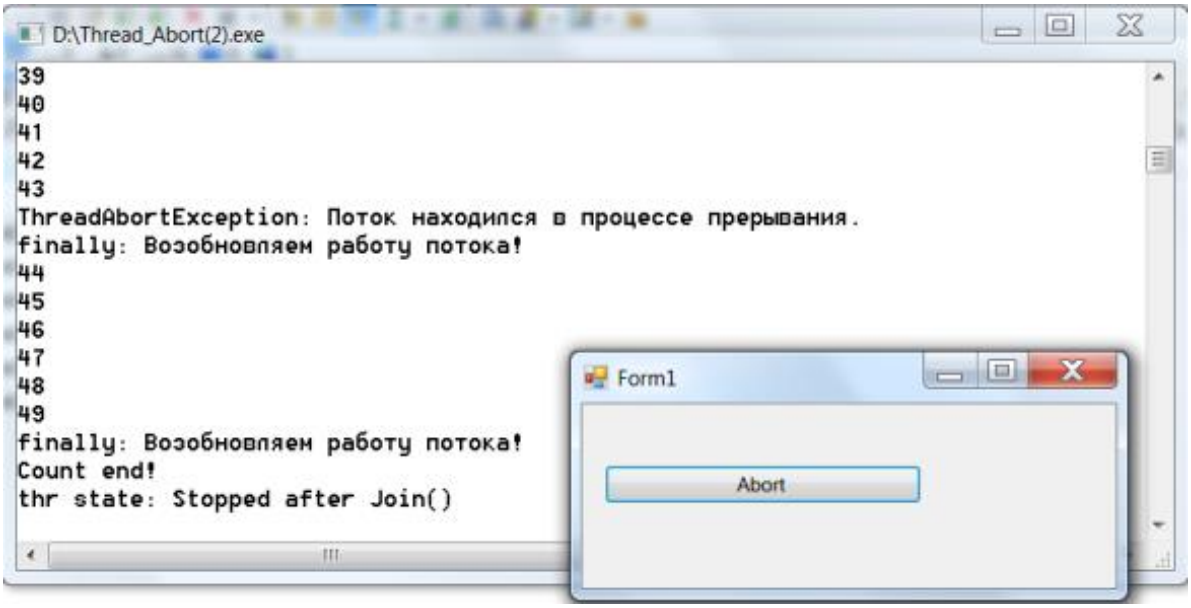

Рис. 3.4. Результат работы приложения

3.2.8. Методы Suspend() и Resume()

Другим способом усыпить поток, отличным отметода *Sleep()* является метод *Suspend()*. Он усыпляет поток на неопределенное время.

Для пробуждения потока можно использовать рассмотренные методы *Interrupt()* и *Abort()*, а также метод *Resume()*, который возобновляет приостановленную работу потока.

Важно отметить, что когда для потока вызывается метод *Suspend()*, он не приостанавливается немедленно. Поэтому не рекомендуется использовать методы Suspend() и Resume() для синхронизации потоков.

4.ЗАДАНИЯ ДЛЯ ЛАБОРАТОРНЫХ РАБОТ

Для выполнения работ могут быть использованы системы программирования MS Visual Studio .Net версий 2012 года и выше.

По результатам выполнения каждой работы студент должен подготовить и представить преподавателю отчет, который включает разделы:

– постановка задачи;

– метод решения задачи;

– описание разработанных классов (структура данных и алгоритмы);

– описание программы;

– тестовый пример;

– результаты, выдаваемые программой;

– выводы.

ЛАБОРАТОРНАЯ РАБОТА 1
Изучение классов пространства *System.Diagnostics* для контроля и диагностики процессов

Цель работы – ознакомление с возможностями классов *System.Diagnostics* для контроля и управления процессами в вычислительной системе.

Задание

1) Изучить назначение и функциональные возможности типов *Process*, *Process*Module, *Process*ModuleCollection, *Process*StartInfo, *Process*Thread пространства *System.Diagnostics*.

2) Получить навыки работы с изучаемыми классами.

Порядок выполнения работы

1. Пункты задания 1–2 изучить по соответствующим разделам данных методических указаний и справочной системе msdn *(https://msdn.microsoft.com/library/system.diagnostics(v=vs.100).aspx).*

2. Пункт 2 задания выполнть на примере решения задачи:

Задача.

Разработать класс для диагностики состояния процессов на локальном компьютере, включающий следующие методы:

1) Получение списка процессов, выполняемых на локальном компьютере.

2) Получение строки описания заданного процесса *(Process*), включающей:

- имя процесса;
- идентификатор процесса;
- имя компьютера;
- время начала процесса.

3) Вывода описания выполняемых на локальном компьютере процессов на консоль.

4) Получение строки описания заданного потока процесса *(ProcessThread*), включающей:

- идентифкатор потока:
- уровень приоритета потока;
- базовый приоритет потока;
- текущий приоритет потока;
- время запуска;
- количество времени, потраченное процессором на выполнение потока.

5) Вывода описания потоков заданного процесса.

6) Получение строки описания заданного модуля процесса *(ProcessModule),* включающей:

- имя модуля;
- полный путь к модулю на диске;
- версию модуля;
- объем памяти, необходимый для загрузки модуля.

Вывода описания модулей заданного процесса.

7) Для запуска процесса выполнения приложения (с использованием и без использования класса *ProcessStartInfo*).

8) Для уничтожения процесса выполнения приложения.

Протестировать разработанный класс, сделать заключение об использованых классах *System.Diagnostics*.

ЛАБОРАТОРНАЯ РАБОТА 2

Сооздание вторичных потоков приложения

Цель работы – освоение различных способов создания вторичных потоков приложения на языке C# для платформы .NET Framework.

Задание

1) Изучить назначение и функциональные возможности типов *Thread, ParameterizedThreadStart, ParameterizedThreadStart* для создания вторичных потоков в приложении на языке C#.

2) Изучить и освоить алгоритм создания вторичного потока в приложении на языке C# с использованием типов *Thread, ParameterizedThreadStart, ParameterizedThreadStart.*

3) Отработать все существующие способы создания потока, следуя заданному порядку выполнения работы.

Порядок выполнения работы

1. Пункты задания 1–2 изучить по соответствующим разделам данных методических указаний.

2. Пункт 3 задания отработать на примере решения задачи:

Задача 1

Вычисления оператора над целочисленными аргументами (допустимые операции: +, –, *, / , %), выполняемого во вторичном потоке в описанном ниже формате.

Задача 2

Вычисления оператора над вещественными аргументами (допустимые операции: +, –, *, / , %), выполняемого во вторичном потоке, в описанням ниже формате.

Формат для задания оператора:

операнд1 операция операнд2.

Для решения задачи метод вычисления для вторичного потока определить всеми указанным способами:

1) как анонимный метод без параметров;
2) как лямбда-выражение без параметров;
3) как объектный метод класса без параметров;
4) как статический метод класса без параметров;
5) как анонимный метод с параметрами, передаваемыми через аргумент типа object;
6) как лямбда-выражение с параметрами, передаваемыми через аргумент типа object;
7) как объектный метод класса с параметрами, передаваемыми через аргумент типа object;
8) как статический метод класса с параметрами, передаваемыми через аргумент типа object;
9) как лямбда-выражение с параметрами, передаваемыми через глобальные параметры;
10) как объектный метод класса с параметрами, передаваемыми через аргумент типа object[].

Оператор для вычисления вводится с консоли. При использовании метода без параметров оператор ввод оператора осуществляется методом потока. При использовании метода с параметрами ввод оператора осуществляется методом, организующим вторичный поток.

Для ожидания завершения вторичного потока использовать метод *Join()*. Каждый реализованный способ оформляется как обработчик нажатия кнопки на форме Windows приложения.

ЛАБОРАТОРНАЯ РАБОТА 3

Исследование приоритетов потоков

Цель работы – изучение свойств класса *Thread* и исследование влияние приоритетов потоков на их активность.

Задание

1) Изучить свойства класса Thread.
2) Изучить влияние приоритета потока на его активность.

3) Проверить на практике влияние приоритета потока на его активность на примере решения предлагаемой задачи.

Порядок выполнения работы

1. Пункты задания 1–2 изучить по соответствующим разделам данных методических указаний.

2. Пункт 3 задания отработать на примере решения задачи:

Задача 1

Разработайте функцию для вывода свойств потока во время его выполнения.

Задача 2

Выполнить пошагово эксперимент 1:

1) Организовать класс *PriorityTesting* для сбора статистики о выполняемых потоках, в котором определить:

- массив счетчиков *long[] counts*;
- флаг завершения работы *bool finish*, значение *true* которого указывает на необходимость завершить метод потока;
- метод для выполнения потока:

```
public void ThreadMetod(object iThread)
    {
       while (true)
       {
          if (finish)
             break;
          counts[(int)iThread]++;
       }
    }
```

2) Метод в качестве параметра получает номер потока *iThread*, в котором он выполняется. Метод выполняет бесконечный цикл, на

каждом шаге которого счетчик потока увеличивается на 1. Таким образом, *counts[i]* после завершения выполнения потока с индексом *i* содержит количество выполненных циклов в методе потока. Выход из цикла осуществляется по состоянию флага *finish.*

3) На форме Windows приложения по нажатию кнопки обеспечить создание пяти потоков с пятью разными приоритетами (*Highest, AboveNormal, Norma, BelowNormal, Lowest*), в каждом потоке выполняется указанный метод *ThreadMetod().*

4) Запустить все потоки на выполнение. Через некоторое время, например, 10 с завершить выполнение потоков, установив флаг *finish* в состояние *true.*

5) Дождаться завершения всех потоков и вывести статистику выполнения потоков в формате:

Имя_потока_ *i* Приоритет_потока_ *i counts[i]*

6) Запустить приложение на выполнение, получить результаты, проанализировать, сформулировать выводы.

Задача 3

Предложите и проведите свой эксперимент 2 для исследования проблемы как влияет приоритет потока на его активность. Проанализируйте и прокомментируйте результаты.

ЛАБОРАТОРНАЯ РАБОТА 4
Изучение возможностей класса *Thread* для управления потоками приложения

Цель работы – освоение различных способов управления потоками в приложении.

Задание

1) Изучить назначение и функциональные возможности методов класса Thread: Sleep(), Join(), Interrupt(), Abort(), ResetAbort(), Suspend(), Resume().

2) Изучить и освоить алгоритм создания вторичного потока в приложении на языке C# с использованием типов *Thread, ParameterizedThreadStart, ParameterizedThreadStart.*

3) Отработать все существующие способы создания потока, следуя заданному порядку выполнения работы.

Порядок выполнения работы

1. Пункты задания 1–2 изучить по соответствующим разделам данных методических указаний.

2. Пункт 3 задания отработать на примере решения задачи:

Задача 1

Разработайте приложение, которое запускает пять потоков, выполняющие один и тот же метод, который в цикле выводит на консоль заданный символ (параметр метода). Исследуйте результат работы приложения в двух случаях:

- после вывода очередного символа вызывается метод *Sleep(0);*
- после вывода очередного символа не вызывается метод *Sleep(0).*

Задача 2

Разработать приложение, моделирующее параллельные вычисления для решения квадратного уравнения*: a*x*x+b*x+c=0* (коэффициенты *a, b, c* вводятся), управляя порядком вычислений методом *Join ():*

1) Определить класс *Calculator*, содержащий методы для выполнения арифметических операций +, –, *, /, которые использовать для организации потоков.

2) В классе *Program* определить метод для вычисления корней уравнения с использованием потоков на основе графа (рис.4.1), вытекающего из метода решения квадратного уравнения.

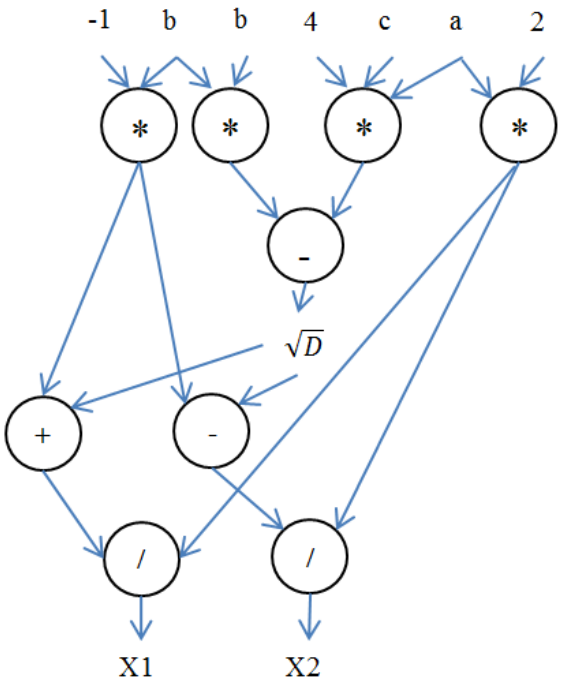

Рис. 4.1. Граф потоков

Задача 3

Разработать приложение, моделирующее работу конвейера обработки данных с накопителями, на вход которого поступают объекты данных из входного файла. После обработки на конвейере объекты результаты записываются в результирующий файл, т.е. каждому исходному объекту данных из входного файла соответствует объект данных результатов в результирующем файле.

Конвейер обработки данных включает *K* ступеней, на каждой из которых выполняется одна операция обработки *OPi (i=1,K)* над каждым объектом данных из входного файла. Каждая ступень имеет накопитель неограниченного размера,работающий в режиме очереди, в котором хранятся данные, ожидающие обработки на этой ступени. Для моделирования конвейера использовать такие правила:

1) Объект данных из входного файла поступает на ступень в очередь накопителя ступени 1.

2) Результат обработки объекта данных ступени *i* является входным объектом данных для ступени *i+1* и поступает в накопитель этой ступени, если $i+1 \leq K$, иначе записывается в результирующий файл.

3) На каждой ступени конвейера может выполняться операция этой ступени над одним объектом данных – первым в очереди накопителя этой ступени.

4) Порядок обработки объектов данных на ступенях конвейера совпадает с исходным порядком объектов данных в файле и не может быть нарушен.

5) Каждая ступень конвейера имеет свою длительность выполнения операции.

6) Конвейер останавливается, если заполнение накопителя первой ступени прекращено, все ступени закончили свои операции обработки и все накопители пусты.

Каждую ступень конвейера моделировать потоком, выполняющим функцию этой ступени с заданной задержкой. Конвейер считается запущенным, если запущены на выполнение все потоки, моделирующие его ступени. Если для потока, моделирующего ступень, в его накопителе есть очередь ожидающих обработки данных, поток их обрабатывает, добавляет результаты в накопитель следующей ступени (или записывает в результирующий файл, если эта ступень последняя).

Если накопитель ступени пуст,то поток переводится в состояние сна, а выводится из него, если предшествующая ступень записывает в его накопитель свой результат.

В процессе моделирования на консоль выводить информацию о переходах ступени в новое состояние с указанием момента времени перехода (отсчёт ведётся от момента первой загрузки ступени 1).

По результатам моделирования построить график Ганта загрузки конвейера, на котором выделять время обработки и простоя.

7) Для решения задачи использовать методы *Sleep(), Interrupt(), Abort(), ResetAbort()*.

8) Для выполненной реализации заменить использованные методы класса *Thread Sleep(), Interrupt()* на *Suspend(), Resume().* Сравнить две реализации и сделатьзаключение по методам *Suspend(), Resume().*

ЛАБОРАТОРНАЯ РАБОТА 5
Использование возможностей класса *Thread* для разработки параллельного приложения

Цель работы – освоение приёмов распараллеливания алгоритмов для решения программистских задач.

Задание

1) Изучить функциональные возможности методов класса *Thread* для создания многопоточных приложений.
2) Изучить приёма распараллеливания алгоритмов для создания параллельных решений.
3) Выполнить решение предложенных для параллельной реализации задач.

Порядок выполнения работы

1. Пункты задания 1–2 изучить по соответствующим разделам данных методических указаний.
2. Пункт 3 задания отработать на примере решения задачи:

Задача 1

Разработать класс для представления объектов данных типа «Вектор» с параллельно реализованными операция:
- сложение веторов;
- вычитание векторов;
- умножение вектора на скаляр;
- модуль вектора.

Задача 2

Разработать класс для представления объектов данных типа «Матрица» с параллельно реализованными операциями:

- сложение матриц;
- вычитание матриц;
- умножение матрицы на скаляр;
- умножение матрицы на вектор;
- умножение вектора на матрицу;
- умножение матрицы на матрицу;
- возведение матрицы в степень;
- вычисление детерминанта матрицы;
- вычисление обратной матрицы;
- вычисление транспонированной матрицы.

Задача 3

Разработать класс для вычисления определенного интеграла функции разными методами, которые реализовать параллельно:

- методом левых прямоугольников;
- методом правых прямоугольников;
- методом трапеций.

Задача 4

Разработать параллельную реализацию нахождения среднего арифметического последовательности чисел с файла.

Вопросы для контроля

1) Дайте определение процесса.
2) Как соотносятся понятия приложение и процесс применительно к платформе .Net?
3) Какие классы пространства System.Diagnostis вы знаете, их назначение и возможности?
4) Перечислите классы для управления процессами.
5) Что понимается под доменом процесса?
6) Какое приложение называется многопоточным?
7) Назначение потока.
8) Опишите организацию потока.
9) Определите назначение и возможности класса *Thread*.
10) Чем отличаются классы *Thread* и *ProcessThread?*
11) Опишите этапы запуска метода на исполнение в потоке.
12) Продемонстрируйте запуск метода класс без параметров на исполнение в потоке.
13) Продемонстрируйте запуск метода класс с параметрами на исполнение в потоке.
14) Продемонстрируйте запуск анонимного метода без параметров на исполнение в потоке.
15) Продемонстрируйте запуск анонимного метода с параметрами на исполнение в потоке.
16) Продемонстрируйте запуск метода без параметров на исполнение в потоке на основе лямбда-выражения.
17) Продемонстрируйте запуск метода с параметрами на исполнение в потоке на основе лямбда-выражения.

18)Как передать исходные данные в метод потока без явной передачи через параметры?

19)Перечислите известные вам свойства класса *Thread*, опишите их.

20)Перечислите известные вам методы класса *Thread*, укажите их функциональное назначение.

21)Какие методы управления потоками класса *Thread* вы знаете?

22)Какие способы усыпления потока вы знаете?

23)Как разбудить поток?

24)Как вызвать завершение потока?

25)Какие исключения выбрасываю методы *Interrupt()* и *Abort()?*

26)Как можно дождаться завершения потока?

27)Какие потоки называются фоновыми?

28)Какой поток называется основным?

29)Как можно отменить завершение потока?

30)Какой выигрыш даёт многопоточное программирование?

СПИСОК ЛИТЕРАТУРЫ

1. Рейли Д. Создание приложений Microsoft ASP.Net / Д.Рейли : пер.с англ.– М.: Изд.-торговый дом «Русская редакция», 2002.– 480с., ил.

2. Петцольд Ч. Программирование для Microsoft Windows на C# : В 2-х т. Т.1. / Ч. Петцольд : пер. с англ. – М.: Издательско-торговый дом «Русская Редакция», 2002.– 576 с., ил.

3. Петцольд Ч. Программирование для Microsoft Windows на C#: В 2-х т. Т.2. / Ч. Петцольд» : пер. с англ. – М.: Издательско-торговый дом «Русская Редакция», 2002.– 624 с., ил.

4. Лабор В. В. Си Шарп : Создание приложений для Windows / В. В. Лабор.– Мн.: Харвест, 2003.– 384 с.

5. Рихтер Дж. Программирование на платформе Microsoft .NET Framework / Дж. Рихтер : пер. с англ. – 2-е изд., испр. – М.: Издательско-торговый дом «Русская Редакция», 2003.– 512 с., ил.

6. Разработка Windows-приложений на Microsoft Visual Basic .NET и Microsoft Visual C# -NET : учеб. курс MCAD/MCSD : пер. с англ. – М.: Издательско-торговый дом «Русская Редакция», 2003.– 512 с., ил.

7. Троелсен Э. C# и платформа NET. Библиотека программиста / Э. Троелсен. – СПб.: Питер, 2003. – 800 с.,ил.

8. Анализ требований и создание архитектуры решений на основе Microsoft .NET : учеб. курс MCSD/пер. с англ.–М.: Издательско-торговый дом «Русская Редакция», 2004.– 416 с., ил.

9. Шилдт Г. Полный справочник по C# / Шилдт Г. : пер. с англ. – М.: Издательский дом "Вильямс", 2004. – 752 с., ил.

10. Бишоп Дж. C# в кратком изложении / Дж.Бишоп, Н.Хорспул : пер. с англ.– М.: Бином, Лаборатория знаний, 2005. – 472 с., ил.

СОДЕРЖАНИЕ

Учет использования методических указаний

№	Ф.И.О. студента	Дата	Роспись

Навчальне видання

Методичні вказівки до лабораторних робіт
«Основы параллельного программирования на языке C#»
з дисципліни «Технології програмування»
для студентів спеціальностей
122 – Компютерні науки та інформаційні технології,
124 – Системний аналіз, у тому числі для іноземних студентів

Російською мовою

Укладачі:

КОЖИН Юрій Миколайович

МАЛИХ Олег Миколайович

ПРОКОПЕНКОВ Володимир Пилипович

Відповідальний за випуск О. С. Куценко

Роботу до друку рекомендував О. В. Горілий

Редактор О. І. Шпільова

План 2017 , поз. 1

Підп. до друку Формат 60x84 1/16 Папір офсетний.

Riso-друк. Гарнітура Таймс. Ум.друк.арк. 2,8

Наклад 100 прим. Зам. № Ціна договірна.

Видавничий центр НТУ “ХПІ”,

61002, м.Харків, вул. Кирпичова, 2

Свідоцтво суб’єкта про реєстрацію ДК №3657 від 24.12.2009 р.

Друкарня НТУ “ХПІ” . 61002, м.Харків, вул. Кирпичова, 2